LESSONS LEARNED FROM PORTING HELENOS TO RISC-V

Martin Děcký
martin@decky.cz

February 2019
Who Am I

- **Passionate programmer and operating systems enthusiast**
  - With a specific inclination towards multiserver microkernels
- **HelenOS developer since 2004**
- **Research Scientist from 2006 to 2018**
  - Charles University (Prague), Distributed Systems Research Group
- **Senior Research Engineer since 2017**
  - Huawei Technologies (Munich), German Research Center, Central Software Institute, OS Kernel Lab
HelenOS in a Nutshell

open source general-purpose multiplatform microkernel multiserver operating system designed and implemented from scratch
HelenOS in a Nutshell

open source general-purpose multiplatform microkernel multiserver operating system designed and implemented from scratch

Custom microkernel
Custom user space
http://www.helenos.org
HelenOS in a Nutshell

open source general-purpose multiplatform microkernel multiserver operating system designed and implemented from scratch

3-clause BSD permissive license
https://github.com/HelenOS
HelenOS in a Nutshell

open source general-purpose multiplatform microkernel multiserver operating system designed and implemented from scratch

Breath-first rather than depth-first
Potentially targeting server, desktop and embedded
HelenOS in a Nutshell

open source general-purpose multiplatform microkernel multiserver operating system designed and implemented from scratch

IA-32 (x86), AMD64 (x86-64), IA-64 (Itaninum), ARM, MIPS, PowerPC, SPARCv9 (UltraSPARC)
HelenOS in a Nutshell

open source general-purpose multiplatform microkernel multiserver operating system designed and implemented from scratch

Fine-grained modular component architecture
No monolithic components even in user space
HelenOS in a Nutshell

open source general-purpose multiplatform microkernel multiserver operating system designed and implemented from scratch

Architecture based on a set of guiding design principles
Asynchronous bi-directional IPC with rich semantics
Motivation: Software Dependability

How HelenOS tries to achieve dependability?

- Microkernel multiserver architecture based on design principles
  - Fundamental fault isolation (limiting the “blast radius”)
  - Explicit mapping between design and implementation
- Clean, manageable, understandable and auditable source code
  - “Code is written once, but read many times”
  - Ratio of comments: 38 %
    - “Extremely well-commented source code” (Open Hub)
- Work in progress: Formal verification
Motivation: Software Dependability

- High-quality architecture
- High-quality implementation
- Verification of correctness
- Development process
monolithic OS

HelenOS
HelenOS Microkernel Functional Blocks

- ELF loader
- kernel lifecycle mgmt
- kernel log
- generic resource allocator
- concurrent hash table
- read-copy update
- system information
- cycle & time mgmt
- tracing support
- misc routines
- work queues
- interrupt & syscall dispatch
- hardware resource mgmt
- string routines
- memory reservation
- wait queues
- thread & task mgmt
- IPC
- slab allocator
- work queues
- thread scheduler
- capabilities
- memory backends
- frame allocator
- spinlocks
- context switching
- platform memory mgmt
- memory zones mgmt
- cache coherency
- platform library routines
- interrupt handling
- platform drivers
- debugging support
- global page hash table support
- shared platform drivers
- bootstrap routines
- CPU mgmt
- context switching
- platform memory mgmt
- atomic & barriers
- shared debugging support
- kernel debug console
- kernel unit tests
- hierarchical page table support
- ELF loader
- kernel lifecycle mgmt
- kernel log
- generic resource allocator
- concurrent hash table
- read-copy update
- system information
- cycle & time mgmt
- tracing support
- misc routines
- work queues
- interrupt & syscall dispatch
- hardware resource mgmt
- string routines
- memory reservation
- wait queues
- thread & task mgmt
- IPC
- slab allocator
- work queues
- thread scheduler
- capabilities
- memory backends
- frame allocator
- spinlocks
- context switching
- platform memory mgmt
- memory zones mgmt
- cache coherency
- platform library routines
- interrupt handling
- platform drivers
- debugging support
- global page hash table support
- shared platform drivers
- bootstrap routines
- CPU mgmt
- context switching
- platform memory mgmt
- atomic & barriers
- shared debugging support
- kernel debug console
- kernel unit tests
- ELF loader
- kernel lifecycle mgmt
- kernel log
- generic resource allocator
- concurrent hash table
- read-copy update
- system information
- cycle & time mgmt
- tracing support
- misc routines
- work queues
- interrupt & syscall dispatch
- hardware resource mgmt
- string routines
- memory reservation
- wait queues
- thread & task mgmt
- IPC
- slab allocator
- work queues
- thread scheduler
- capabilities
- memory backends
- frame allocator
- spinlocks
- context switching
- platform memory mgmt
- memory zones mgmt
- cache coherency
- platform library routines
- interrupt handling
- platform drivers
- debugging support
- global page hash table support
- shared platform drivers
- bootstrap routines
- CPU mgmt
- context switching
- platform memory mgmt
- atomic & barriers
- shared debugging support
Lessons Learned from Porting HelenOS to RISC-V
HelenOS RISC-V Port Status

- **January 2016**
  - Infrastructure, boot loader, initial virtual memory setup, kernel hand-off
    - Privileged ISA Specification version 1.7, toolchain support not upstreamed yet
    - Targeting Spike
    - 18 hours net development time
  - Initial experience
    - Many things besides the ISA itself were not nicely documented (e.g. ABI, HTIF) and had to be reverse-engineered from Spike
    - Even some ISA details were sketchy (memory consistency model)
    - Generally speaking, the ISA itself looked nice (except the compressed page protection field)
August 2017

- Basic kernel functionality (interrupt/exception handling, context switching, atomics, basic I/O)
  - Privileged ISA Specification version 1.10
    - Some minor improvements (e.g. more standard page protection bits)
  - Still targeting Spike
    - Observation: The HTIF input device has a horrible design
      - No interrupts
      - Polling requests are buffered
    - Still no decent “reference platform”
- 24 hours net development time
test/synch/rcu1.c: In function 'seq_func':
test/synch/rcu1.c:446:1: error: unrecognized insn:
} ~

(insn 197 43 47 7 (set (mem/c:SI (symbol_ref:DI ("seq_test_result") [flags 0x86] <var _decl 0x7fe984630240 seq_test_result>)) [12 seq_test_result+0 S4 A32])
  (reg:SI 156)) "test/synch/rcu1.c":413 -1
  (nil))
test/synch/rcu1.c:446:1: internal compiler error: in extract_insn, at recog.c:2311
0x8dd483 _fatalInsn(char const*, rtx_def const*, char const*, int, char const*)
/root/install/cross/riscv64/gcc-7.1.0/gcc/rtl-error.c:108
0x8dd4b9 _fatal_insn_not_found(rtx_def const*, char const*, int, char const*)
/root/install/cross/riscv64/gcc-7.1.0/gcc/rtl-error.c:116
0x8b3371 extract_insn(rtx_insn*)
/root/install/cross/riscv64/gcc-7.1.0/gcc/recog.c:2311
0xd624f3 get_implicit_reg_pending_clobbers(unsigned long (*) [2], rtx_insn*)
/root/install/cross/riscv64/gcc-7.1.0/gcc/sched-deps.c:2871
January 2019

Towards user space support
- Switching to QEMU virt target
  - Looks more reasonable than Spike
  - CLINT, PLIC, NS16550 UART, VirtIO
- Toolchain support upstream
- 8 hours net development time
Lessons Learned

- Suprisingly little interest in porting HelenOS to RISC-V
  - Compared to previous porting efforts to ARM, SPARCv9, SPARCv8, etc.
  - GSoC, master thesis, team software project to no avail
  - Possible reasons
    - Lack of feature-rich reference platform
    - Lack of easily available development board
      - A Raspberry Pi (USB, ethernet, HDMI, sound), but with a RISC-V CPU supporting the Supervisor mode
  - Despite RISC-V being a new major ISA, there is surprisingly little input from operating system research
Problem Statement

- **Microkernel design ideas go as back as 1969**
  - RC 4000 Multiprogramming System nucleus (Per Brinch Hansen)
    - Isolation of unprivileged processes, inter-process communication, hierarchical control
  - There are obvious benefits of the design for safety, security, dependability, formal verification, etc.

- **Hardware and software used to be designed independently**
  - Designing CPUs used to be an extremely complicated and costly process
  - Operating systems used to be written after the CPUs were designed
  - Hardware designs used to be rather conservative
“While intuitive, the benefits of the small TCB have not been quantified to date. We address this by a study of critical Linux CVEs, where we examine whether they would be prevented or mitigated by a microkernel-based design. We find that almost all exploits are at least mitigated to less than critical severity, and 40% completely eliminated by an OS design based on a verified microkernel, such as seL4.”
HelenOS IPC Example

client

VFS

naming service

tmpfs

naming service

HelenOS IPC Example

client

VFS

naming service

tmpfs

naming service

HelenOS IPC Example

client

VFS

naming service

tmpfs

naming service

HelenOS IPC Example

client

VFS

naming service

tmpfs

naming service
Where RISC-V Could Really Help?

- Mainstream ISAs used to be designed in a rather conservative way
  - Can you name some really revolutionary ISA features since *IBM System/370 Advanced Function*?
  - Requirements on the new ISAs usually follow the needs of the mainstream operating systems running on the past ISAs

- No wonder microkernels suffer performance penalties compared to monolithic systems
  - The more fine-grained the architecture, the more penalties it suffers
  - Let us design the hardware with microkernels in mind!
ANY IDEAS?
Communication between Address Spaces

- **Control and data flow between subsystems**
  - Monolithic kernel
    - Function calls
      - Passing arguments in registers and on the stack
      - Passing direct pointers to memory structures
  - Multiserver microkernel
    - IPC via microkernel syscalls
      - Passing arguments in a subset of registers
      - Privilege level switch, address space switch
      - Scheduling (in case of asynchronous IPC)
      - Data copying or memory sharing with page granularity
Is the kernel round-trip of the IPC necessary?

- Suggestion for synchronous IPC: Extended *Jump/Call* and *Return* instructions that also switch the address space
  - Communicating parties identified by a “call gate” (capability) containing the target address space and the PC of the IPC handler (implicit for return)
    - Call gates stored in a TLB-like hardware cache (CLB)
    - CLB populated by the microkernel similarly to TLB-only memory management architecture
- Suggestion for asynchronous IPC: Using CPU cache lines as the buffers for the messages
  - *Async Jump/Call, Async Return* and *Async Receive* instructions
  - Using the CPU cache like an extended register stack engine
Bulk data

- Observation: Memory sharing is actually quite efficient for large amounts of data (multiple pages)
  - Overhead is caused primarily by creating and tearing down the shared pages
  - Data needs to be page-aligned
- Sub-page granularity and dynamic data structures
  - Suggestion: Using CPU cache lines as shared buffers
    - Much finer granularity than pages (typically 64 to 128 bytes)
    - A separate virtual-to-cache mapping mechanism before the standard virtual-to-physical mapping
Fast Context Switching

- Current microsecond-scale latency hiding mechanisms
  - Hardware multi-threading
    - Effective
    - Does not scale beyond a few threads
  - Operating system context switching
    - Scales for any thread count
    - Too slow (order of 10 µs)

- Goal: Finding a sweet spot between the two mechanisms
Fast Context Switching (2)

- **Suggestion: Hardware cache for contexts**
  - Again, similar mechanism to TLB-only memory management
  - Dedicated instructions for context store, context restore, context switch, context save, context load
    - Context data could be potentially ABI-optimized
  - Autonomous mechanism for event-triggered context switch (e.g. external interrupt)
  - Efficient hardware mechanism for latency hiding
    - The equivalent of fine/coarse-grained simultaneous multithreading
      - The software scheduler is in charge of setting the scheduler policy
      - The CPU is in charge of scheduling the contexts based on ALU, cache and other resource availability
User Space Interrupt Processing

- **Extension of the fast context switching mechanism**
  - Efficient delivery of interrupt events to user space device drivers
    - Without the routine microkernel intervention
  - An interrupt could be directly handled by a preconfigured hardware context in user space
    - A clear path towards moving even the timer interrupt handler and the scheduler from kernel space to user space
    - Going back to interrupt-driven handling of peripherals with extreme low latency requirements (instead of polling)
  - The usual pain point: Level-triggered interrupts
    - Some coordination with the platform interrupt controller is probably needed to automatically mask the interrupt source
Capabilities as First-Class Entities

- **Capabilities as unforgeable object identifiers**
  - But eventually each access to an object needs to be bound-checked and translated into the (flat) virtual address space
  - Suggestion: Embedding the capability reference in pointers
    - RV128 could provide 64 bits for the capability reference and 64 bits for object offset
      - 128-bit flat pointers are probably useless anyway
    - Besides the (somewhat narrow) use in the microkernel, this could be useful for other purposes
      - Simplifying the implementation of managed languages’ VMs
      - Working with multiple virtual address spaces at once
Prior Art

  - Offloading basic microkernel operations (e.g. thread creation, context switching) to hardware shown to improve performance by 15 % on average and up to 73 %
    - This was a coarse-grained approach

- Hardware message passing in Intel SCC and Tilera TILE-G64/TILE-Pro64
  - Asynchronous message passing with tight software integration
Prior Art (2)

  - Practical programming model for using multiple virtual address spaces on commodity hardware (evaluated on DragonFly BSD and Barrelish)
    - Useful for data-centric applications for sharing large amounts of memory between processes

- Intel IA-32 Task State Segment (TSS)
  - Hardware-based context switching
  - Historically, it has been used by Linux
    - The primary reason for removal was not performance, but portability
Prior Art (3)

- **Intel VT-x VM Functions (VMFUNC)**
  - Efficient cross-VM function calls
  - Switching the EPT and passing register arguments
  - Current implementation limited to 512 entry points
  - Practically usable even for very fine-grained virtualization with the granularity of individual functions
      - “The cost of a VMFUNC is similar with a syscall”
      - “… hypervisor-level protection at the cost of system calls”
  - **SkyBridge paper to appear at EuroSys 2019**
Prior Art (4)


  - Hardware-based capability model for byte-granularity memory protection
  - Extension of the 64-bit MIPS ISA
    - Evaluated on an extended MIPS R4000 FPGA soft-core
    - 32 capability registers (256 bits)
  - Limitation: Inflexible design mostly due to the tight backward compatibility with a 64-bit ISA

- Intel MPX

  - Several design and implementation issues, deemed not production-ready
Summary

- Traditionally, hardware has not been designed to accommodate the requirements of microkernel multiserver operating systems
  - Microkernels thus suffer performance penalties
    - This prevented them from replacing monolithic operating systems and closed the vicious cycle
- Co-designing the hardware and software might help us gain the benefits of the microkernel multiserver design with no performance penalties
  - However, it requires some out-of-the-box thinking
- RISC-V has “once in the lifetime” opportunity to reshape the entire computer industry
  - Finally moving from unsafe and insecure monolithic systems to microkernels
Acknowledgements

- OS Kernel Lab at Huawei Technologies
  - Javier Picorel
  - Haibo Chen
Focusing on microkernel research, design and development

- Basic research
- Applied research
- Prototype development
- Collaboration with academia and other technology companies

Looking for senior operating system researchers, designers, developers and experts

- Previous microkernel experience is a big plus
- “A startup within a large company”
- Shaping the future product portfolio of Huawei
  - Including hardware/software co-design via HiSilicon
Q&A
THANK YOU!